Introduction to NaturalX

This section covers the following topics:

- Why NaturalX?
- Programming Techniques

Why NaturalX?

Software applications that are based on component architecture offer many advantages over traditional designs. These include the following:

- Faster development. Programmers can build applications faster by assembling software from prebuilt components.
- Reduced development costs. Having a common set of interfaces for programs means less work integrating the components into complete solutions.
- Improved flexibility. It is easier to customize software for different departments within a company by just changing some of the components that constitute the application.
- Reduced maintenance costs. In the case of an upgrade, it is often sufficient to change some of the components instead of having to modify the entire application.
- Easier distribution. Components encapsulate data structures and functionality in distributable units.

NaturalX enables you to create and distribute object-based applications. Using Distributed Object technology (currently DCOM), it enables you to:

- allow your components to be accessed by other components,
- execute these components on local and/or remote servers,
- access components written in a variety of programming languages across process and machine boundaries from within Natural programs,
- provide your existing Natural applications with (quasi) standardized interfaces.

The following scenario illustrates how a company could exploit these advantages. A company introduces a new sales management system that is based on an application design using components. There are numerous data entry components in the application, one for each sales point. But all of these sales point use a common tax calculation component that runs on a server. If the tax legislation is changed, then only the tax component has to be updated instead of changing the data entry components at each site. In addition, the life of the programmers is made easier because they do not have to worry about network programming, operating-system compatibility, and the integration of components that are written in different languages.

Programming Techniques

This section covers the following topics:

- Object-Based Programming
- Defining Classes
- Defining Interfaces
- Interface Inheritance

Object-Based Programming

NaturalX follows an object-based programming approach. Characteristic for this approach is the encapsulation of data structures with the corresponding functionality into classes. Encapsulation is a good basis for easy distribution. Because there are now (quasi) standards for the interoperation of software components on the basis of object models, an object-based approach is also a good basis for making software components interoperable across program, machine and programming language boundaries.

Defining Classes

In an object-based application, each function is considered to be a service that is provided by an object. Each object belongs to a class. Clients use the services either to perform a business task or to build even more complex services and to provide these to other clients. Hence the basic step in creating an application with NaturalX is to define the classes that form the application. In many cases, the classes simply correspond to the real things that the application in question deals with, for example, bank accounts, aircraft, shipments etc. There is a wide range of good literature about object-oriented design, and a number of well-proven methods can be used to identify the classes in a given business.

The process of defining a class can be broadly broken down into the following steps:

- Create a Natural module of type class.
- Specify the name of the class using the DEFINE CLASS statement. This name will be used by the clients to create objects of that class.
- Use the OBJECT clause of the DEFINE DATA statement to define how an object of the class will look internally. Create a local data area that describes the layout of the object with the data area editor, and assign this data area in the OBJECT clause.

These steps are described in more detail in the section Developing Object-Based Natural Applications.

Defining Interfaces

In order to be useful to clients, a class must provide services, which it does through interfaces. An interface is a collection of methods and properties. A method is a function that an object of the class can perform when requested by a client. A property is an attribute of an object that a client can retrieve or change. A client accesses the services by creating an object of the class and using the methods and properties of its interfaces.

The process of defining an interface can be broadly broken down into the following steps:

- Use the INTERFACE clause to specify an interface name.
- Define the properties of the interface with PROPERTY definitions.
- Define the methods of the interface with METHOD definitions.

These steps are described in more detail in the section Developing Object-Based Natural Applications.

Simple classes only have one interface, but a class may have more than one interface. This possibility can be used to group methods and properties into one interface that belong to the same functional aspect of the class and to define different interfaces to handle other functional aspects. For example, an *Employee* class could have an interface *Administration* that contains all of the methods and properties of the administrative aspects of an employee. This interface could contain the properties *Salary* and *Department* and the method *TransferToDepartment*. Another interface *Qualifications* could contain the qualification aspects of an employee.

Interface Inheritance

Defining several interfaces for a class is the first step towards using interface inheritance, which is a more advanced method of designing classes and interfaces. This makes it possible to reuse the same interface definition in different classes. Assume that there is a class *Manager*, which is to be treated in the same way as the class *Employee* with respect to qualification, but which is to be handled differently as far as administration is concerned. This can be achieved by having the *Qualification* interface in both classes. This has the advantage that a client that uses the *Qualification* interface on a given object does not have to check explicitly whether the object represents an *Employee* or a *Manager*. It can simply use the same methods and properties without having to know of what class the object is. The properties or methods can even be implemented in a different way in both classes provided they are presented through the same interface definition.

The process of using interface inheritance can be broadly broken down into the following steps:

- Use the INTERFACE statements to define one or more interfaces in a copycode instead of defining them directly in the class.
- The METHOD and PROPERTY definitions in the INTERFACE statement do not need to contain the IS clause. At this point, you just define the external appearance of the interface without assigning implementations to the methods and properties.
- Use the INTERFACE clause to include the copycode with its interface definition in each class that will implement the interface.
- Use the METHOD and PROPERTY statements to assign implementations to the methods and properties of the interface in each class that will implement the interface.